VISUS Visualization Research Center

Bachelorarbeit Nr. 253

Real-time Ray Tracing of Volumetric Data

Marcus Richter

Course of Study:	Computer Science
Examiner:	Prof. Dr. Thomas Ertl
Supervisor:	Dipl.-Inf. Michael Krone, Dr. Steffen Frey, M.Sc. John E. Stone
Commenced:	August 31, 2015
Completed:	March 1, 2016
CR-Classification:	I.3.3, I.3.7, I.4.10

Abstract

As hardware development advances, ray tracing becomes more and more viable for real-time. Even ray tracing for volumetric data is possible with interactive frame rates. The molecular visualization program VMD is about to be extended to use volumetric ray tracing to enhance the image quality. In this thesis I show the implementation of a volumetric ray tracer that achieves interactive frame rates. The NVIDIA OptiX framework is used as the foundation for the ray tracing. OptiX is a programmable ray tracing framework designed to help developers to build ray tracing applications. Volumetric ray tracing is implemented using direct volume rendering via ray marching. The problems of intersecting geometry and translucency are solved by casting further rays and good image quality is achieved using a local approximation for ambient occlusion. The results show that interactive frame rates are possible for standard desktop PCs. But with activated ambient occlusion only offline rendering can be used.

Kurzfassung

Mit der fortschreitenden Hardware-Entwicklung wird es immer prktikable Raytracing in Echtzeit durchzuf"uhren. Auch für die volumetrische Daten ist es möglich, interaktiven Bildraten zu erzeugen. Das molekulare Visualisierungsprogramm VMD soll erweitert werden volumetrisches Raytracing zu verwenden, um die Bildqualität zu verbessern. In dieser Arbeit zeige Ich die Implementierung eines volumetrischen Raytracer, welcher interaktive Bildraten erreicht. Das NVIDIA OptiX Framework wird als Grundlage für das Raytracing eingesetzt. OptiX ist ein programmierbarsr Raytracing Framework, welches um Entwicklern hilft, Raytracing-Anwendungen zu erstellen. Volumetrisches Raytracing wurde "uber das direkte Volumen Rendering Verfahren namens Ray Marching implementiert. Die Probleme von "uberschneudender Geometrie und Transparanez wurden durch das Erzeugen weiterer Strahlen gelöst und eine gute Bildqualität wurde durch eine lokale Näherung für Ambient Occlusion erreicht. Die Ergebnisse zeigen, dass für Standard-Desktop-PCs interaktive Bildraten m"oglich sind. Mit aktiviertem Ambient Occlusion kann jedoch nur offline Rendering verwendet werden.

Contents

1 Introduction

The open source program VMD is under active development at the University of Illinois and can display, animate, and analyze large biomolecular systems using 3-D graphics and built-in scripting [Pag08]. Molecular data sets usually store the positions and elements of all atoms that form a molecule or a molecular system. Typical representations for biomolecules like proteins, are stick representations, space filling models, cartoon representations, or molecular surfaces. These models can either be rendered using explicit descriptions of the geometry (e.g., a sphere), or classical polygonal 3D graphics [HDS96].

In addition to these geometric models, however, volumetric data like cryo-EM maps or electrostatic potential maps can also be important for data analysis. In this thesis I describe a direct volume rendering system that I implemented, that can later be added to VMD for that purpose. A real-time ray tracer for volumetric data should be implemented to achieve the best possible image quality. To build the ray tracer Nvidia OptiX should be used, which is a programmable ray tracing framework that runs on Nvidia GPU's for real-time performance.

1.1 Goals for the Ray Tracer

The final ray tracer should be able to display volumetric data and support the combination with 3D triangle geometry. With the implementation, interactive frame rates for medium-sized input data with 512^3 voxel resolution and below should be reached. The project should be integrated either into VMD or in a standalone prototype. Intersection between volumes and other geometry should be supported and translucency should be displayed correctly. It should offer good image quality that is achieved by advanced lighting techniques such as Ambient Occlusion.

1.2 Structure

The thesis is structured as follows:

Kapitel 2 – Fundamentals: Here the basic fundamentals are explained that are needed for the understanding of the rest of this thesis. In the first section the basic idea of ray tracing is explained and in the second section an introduction to the OptiX ray tracing framework is given.

Kapitel 3 – Related Work: Previous work related to VMD, volumetric ray tracing and NVIDIA OptiX is presented.

Kapitel 4 – Implementation: In this chapter my approach of solving the problems and the implementation is explained in detail. I explain the project setup and the planned structure of the software. Then the implementation of the ray marching is presented and I show how certain problems like the intersection of objects and translucent geometry is solved. Finally I explain my implementation of ambient occlusion to enhance the visual quality.

Kapitel 5 – Results Shows the results of my implementation and gives a brief summary. First the visual results and the performance are discussed, then the limitations are stated and how the work could be extended in the future. Finally I give a brief summary.

2 Fundamentals

2.1 Introduction to Volumetric Ray Tracing

Ray tracing is one of many techniques to render images with computers. It is mostly used to achieve good image quality in offline rendering as the scene in Fig. 2.1 shows. The achievable image quality is very high and the concept is rather simple, since all it comes down to, is finding the intersection of a line with an object and then shade the point of intersection [Kuc87]. It is a physically very correct approximation and therefore effects like shadows come also physically correct and almost automatically with the method. It is mostly used in off-line rendering for its good image quality, but because the complexity scales with the resolution and the amount of geometry, it is very expensive compared to other methods. Therefore in real-time rendering where we have large hardware constraints, rasterization can achieve visually better looking results, if not physically correct.

The idea behind ray tracing is to create images by trying to calculate the travel of light. Light is emitted by light sources, bounces around the scene and eventually hits the eye or the lens of the camera [Gla89]. By simulating the light it would be possible to determine what the camera sees. But computers are limited to what we can calculate and correct calculation of light does not map well to discretization. When light hits a

Figure 2.1: Scene rendered using ray tracing to achieve high image quality (https://en.wikipedia.org/wiki/Ray_tracing_%28graphics%29)

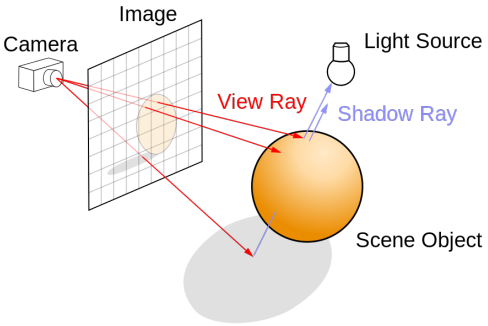

Figure 2.2: Ray tracing uses one ray per pixel (red). Secondary rays (blue) are used for light calculations and other effects.(https://en.wikipedia.org/wiki/Ray_tracing_%28graphics%29)

diffuse surface for example, the light gets partially reflected in all directions. This implies all calculations have to use integrals, which can only be approximated in computer science. In ray tracing, the light instead is modeled as light rays, which is an often used abstraction in physics. A ray is one sample of the light and can be seen as the path of multiple parallel photons traveling through the scene. In stochastic ray tracing [PH10] multiple rays are used to achieve an approximation of the light. The more rays used, the better the approximation. For the best possible approximation with limited calculation power, it is therefore important to choose the right rays, to achieve the best result with the least amount of rays possible.

From all the light that comes from a light source only a very small fraction of it really ends up hitting the lens of the camera. So casting rays from the light source and tracing all of them would mean that most of the calculated rays are of no use since only the ones that hit the camera are important. Instead a backwards approach is used, that spawns rays from the camera and searches for the path light could get into the lens from a specific direction. Because a pixel is only a single color, we need to sample only one ray per pixel, except for multi-sampling used in anti-aliasing. In Fig. 2.2 a visualization of the ray tracing approach is shown. This way, only rays are calculated that we are interested in and hit the camera. To calculate the color and the intensity of a ray that comes from one point on a surface, it is needed to accumulate all light that hits this exact point from all directions. So again here we need discretization and therefore sample-rays are recursively sent in random directions to approximate the result. As light may possibly bounce around the scene indefinitely, an exit condition like a maximum recursive depth may be used.

To determine what geometric object a ray hits, a ray intersection test has to be done for all objects in the scene. This can be greatly accelerated for example using uniform grids or spatial hierarchies to divide space and allow for faster search algorithms. Another popular example for such an Acceleration Structure is the bounding volume hierarchy

method, that uses axis aligned bounding boxes to divide and subdivide the space into a hierarchy of bounding boxes [WBS07]. Then the hierarchy is used like a search tree where the ray is first tested against the largest bounding boxes to quickly cull large regions of space. Other acceleration techniques for example try to minimize the amount of rays like early ray termination where rays are terminated if the accumulated color for example is already white [Ize09].

Volumetric ray tracing raises new challenges because volumetric data does not deliver clearly defined surfaces that the rays can hit. The space between two points in the scene may not be empty and therefore can not be just skipped. A common modification to ray tracing is ray marching [Wei06]. The idea is to march along the ray with a fixed step size and take color samples. So the space between two points is not skipped and instead stepped through and sampled. Since volumetric data often times is also discrete and stored in 3d voxel textures, the step size can be set to the voxel size, and as a result the sampling does not mean any loss of accuracy.

Every ray is an independent instance of the ray tracing problem. So ray tracing should map very well to parallelization and the current GPU technology. However because of the better performance, rasterization is still used in most graphic applications. For this reason the regular GPU hardware for desktop PCs is very specific to rasterization. Ray tracing therefore requires special modifications, to use hardware acceleration from GPUs, which makes it even less efficient. But as parallelization becomes more and more important in the recent development of computer science, GPUs become increasingly used for non graphical scenarios. As a result GPUs are designed to be less specific to rasterization so they can be used in other areas to speed up calculations of all sort. Without hardware acceleration ray tracing is barely feasible for real-time. But Nvidia OptiX uses the GPU hardware to the best extent to make it possible.

2.2 Introduction to the OptiX Framework

The Nvidia OptiX API [PBD+10] uses the computational power of Nvidia GPUs, to provide a powerful yet easy to learn and understand ray tracing API. It offers a full framework for typical ray tracing tasks, so that the programmer does not have to care about the setup of a ray tracing engine and can instead fully concentrate on the core of his software.

What OptiX provides is not a fully fledged ray tracing engine but rather a framework for building ray tracing engines. Therefore almost all important tasks, such as ray generation or intersection tests, still need to be implemented by the user and fed to OptiX [15b]. This way it is ensured to offer the most freedom possible to use the framework for many

different needs. OptiX for example may also be used for non graphic purposes such as calculation of sound propagation or collision detection in a physics engine.

The framework is mostly separated in two parts [15a]. 1) The host-based API that defines ray tracing based data structures and 2) a CUDA C-based programming language where the user can produce new rays, intersect rays with surfaces, and respond to those intersections.

The host API is an object-based C API. Since object orientation was clearly in mind in the design, OptiX also provides a C++ interface to operate all tasks of the host API. This offers a more natural object oriented experience.

In the next section i will present the main objects used in OptiX that are important for this thesis.

2.2.1 OptiX Objects

Context

This is the most top level object of the host API. It serves as an instance of a running OptiX engine. While multiple OptiX instances can be active at once, in most cases it is unnecessary because in a single Context instance, multiple ray tracing tasks can be performed, and even multiple hardware devices can be used. All of the other objects listed below are created by the Context, for example with the call `context->createGroup`. The computation of the ray tracing engine can be invoked with `context->launch`.

Geometry

A Geometry object represents one or more primitives that a ray can be intersected with, such as triangles or other user-defined types. It needs two programs attached to be valid (for more information on the programs see section 2.2.2). Since multiple primitives may be represented by one Geometry object, both these programs will be given the primitive index as an argument by the OptiX engine. The first program is an Intersection Program that must calculate whether a ray hits the primitive. This implicitly defines the shape of the Geometry. In the case of a triangle mesh geometry for example, the geometry object could hold buffers for storing the triangles, then the Intersection Program will test if the triangle at the given index was hit by the ray.

Secondly a Bounding Box Program must be assigned to the Geometry object. It must return an axis aligned bounding box that completely encloses the primitive at the given index. This program is used by the Acceleration Structure of OptiX.

Material

A Material defines what has to be done if a geometry was hit by a ray. In terms of graphical applications it can be seen as surface shader. Two types of programs may be assigned to a Material, a Closest Hit Program and ans Any Hit Program. The first one will be invoked at most once per ray, for the closest intersection of a ray with the primitive. It typically performs shading tasks like texture lookups, reflectance color computations light source sampling and so on. It can also spawn new rays, which is typically used for shadowing or invoking a recursion for calculating reflection.

The Any Hit Program is called on all primitives the ray intersects during traversal. It can terminate the ray, which may for example be used for shadow rays that only need to determine whether the ray hits or not, and it does not matter what was hit.

In OptiX one can define multiple ray types. It is important to notice that the Closest Hit and Any Hit Program may be defined per ray type, which means one Material can hold more than one of these programs. This is useful as different rays need to perform different actions.

Variable

A name used to pass data from the host to the OptiX programs or for communication between programs. Variables are always bound to a Program, Geometry, Geometry Instance, Material or Context object. They can only be written by the host but not read. For communication from the device to the host, OptiX Buffers should be used instead.

After being declared via `rtDeclareVariable`, either by the host or inside a Program, they can be written and read inside Programs. What Variables are visible in what Programs can be seen in table 2.1. If Variables with the same name exist across different objects, the standard rules of scoping apply. Table 2.1 shows the scope search order from left to right. For example a Closest Hit Program that refers to a Variable will search the Program, Geometry Instance, Material and then the Context for the definition.

Buffer

A multidimensional array that can be bound to a variable. The Buffer is stored on the GPU memory and can be attached via the bound variable to objects. For example a Geometry object may have a Buffer for storing vertex data and a Material may use Buffers for textures. Buffers can also be used to communicate between the host and the device, since it is possible to read and write from both sides.

Texture Sampler

One or more Buffers can be bound to a Texture Sampler to offer interpolation

Ray Generation	Program	Context		
Closest Hit	Program	GeometryInstance	Material	Context
Any Hit	Program	GeometryInstance	Material	Context
Intersection	Program	GeometryInstance	Geometry	Context
Bounding Box	Program	GeometryInstance	Geometry	Context
Miss	Program	Context		

Table 2.1: Scope search order for each type of program (from left to right)

mechanisms. It can be bound to a Variable subsequently to other objects such as Materials. This is especially useful for textures.

Geometry Instance

A Geometry always needs a Material to have shading programs that can be invoked once the Geometry is hit in ray traversal. This combination of a Geometry and a Material is called a Geometry Instance. Multiple Materials are also possible in one instance, to support primitives of several different Materials flattened into a single geometry object. The same Materials and Geometries can be used in different instances. For example a sphere geometry type bound to two different instances that use different Materials were possible, to get two spheres with different shading in the scene. Or a single Material can be used for multiple different Geometry objects.

Geometry Group

OptiX uses a graph that is traversed in the tracing of a ray. The user can freely build the graph to his needs. A sample graph is shown in Fig. 2.3.

A Geometry Group object groups multiple Geometry Instances together. This grouping can be used in conjunction with Transform nodes to transform multiple Geometry Instances at once. A Geometry Instance can be part of multiple Geometry Groups, to allow advanced grouping for transformations. Then the Group typically should not be added to the main graph used for traversal. But doing so allows to copy Geometry Instances in the scene and place them at different positions via Transforms.

Group

A higher level grouping object to arrange objects in a hierarchy. While in Geometry Groups only Geometry Instances are allowed as children, a Group can have Geometry Groups, Transforms and Groups themselves as children. This allows grouping of groups to build a more complex hierarchy.

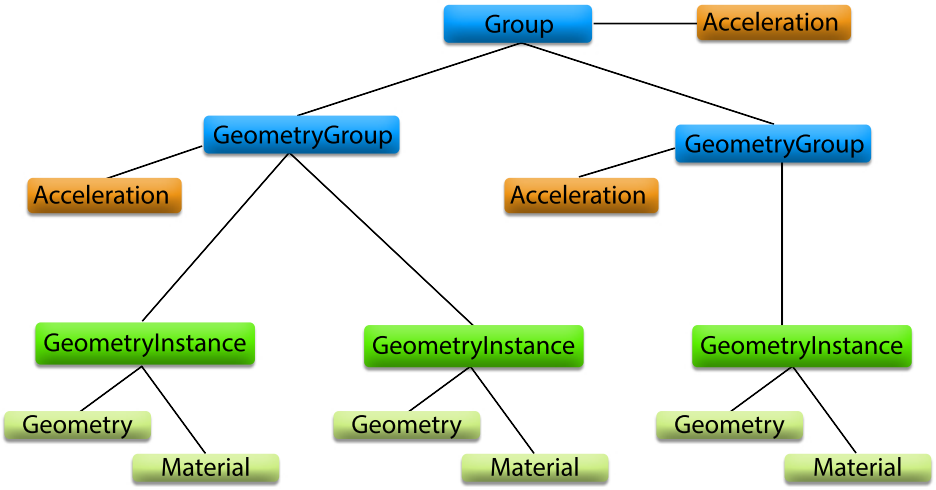

Figure 2.3: Sample graph of a typical OptiX hierarchy

Transform

A Transformation that can be added to the hierarchy, to transform all underlying nodes accordingly. For performance reason, rather then actually transforming the geometric objects of the underlying nodes, the rays are being transformed once a ray encounters a Transform node in the ray traversal. This will become important because when operating in the different Programs such as the Closest Hit or the Any Hit Program, the current ray might be either in object space with transformations applied, or in world space without the transformations.

Acceleration

Acceleration Structures are important for speeding up the ray traversal. They build spatial hierarchies and use traversers for efficient searching in these hierarchies. OptiX offers a variety of different building methods that differ in the performance of building and the quality of the resulting acceleration tree structure. Depending on the scene and other factors some methods might be more suited then others. To balance the trade-offs between the different building methods it is even use different types within the same graph.

For some building methods one can also choose between multiple traversers that will later be used to traverse the tree to find intersection. Again the different types of traversers have trade-offs depending on the needs.

The user cannot create his own Acceleration Structures so one can only choose between the ones given. As those are well implemented, highly efficient and well tested, in most cases they are the best choice anyway.

Every Group and Geometry Group needs to have an Acceleration object assigned. Before an OptiX Context is launched which starts the ray tracing computation, the context must be compiled. In this stage the Acceleration Structures are compiled by building their acceleration trees. If the scene changes between context launches, the context needs to be recompiled. Transformation changes of a transformation node do not require a recompile since they transform the rays during traversal.

Program
A function in Nvidia's PTX virtual assembly language and attached to a Program object. Such program objects are used to assign functions to other objects, for example a geometry object needs a function for intersection testing assigned.

Programs can be written as a function in the CUDA C language. Then the CUDA file needs to be compiled to PTX via the CUDA compiler before runtime.

2.2.2 OptiX Programs

Here I introduce to the different programs used throughout OptiX and explain the inputs and expected outputs.

Ray Generation Program
The Ray Generation Program is the first point of entry after the context was launched. It can be seen similar to a `main` function in a C program. Typically it is used to cast the rays that come from the camera.

To start a new trace `rtTrace` must be called with the top object of the hierarchy that shall be traced, the ray with its origin, direction and valid range, as well as the payload. The payload is an object that will be carried through the trace along with the ray. This allows to store all informations about the trace and its results in the payload. From there, the Ray Generation Program can read it after the trace is finished and use the results for example to write an output buffer.

Intersection Program
Intersection Programs must be assigned to geometry objects. This program must calculate the intersection between a ray and a primitive from the geometry. It is used by the OptiX engine during traversal do the hit testing. The Intersection Program may not be called on all primitives, as the Acceleration Structure may cull regions.

Because a geometry object may consist of more than one primitive, the intersection program receives the primitive index as an argument to identify what primitive should be tested against. `rtCurrentRay` gives access to the ray for testing the

intersection. To accommodate for transformations, the returned ray is in object space, which means that all the transformations that should apply to the geometry, have been inversely applied to the ray instead.

If an intersection is found to be true, the intersection program must first call `rtPotentialIntersection` with the parametric t-value as the argument. `rtPotentialIntersection` returns `true` if the intersection lies inside the allowed range of the ray, otherwise `false`. If it returns `true`, afterwards `rtReportIntersection` must be called with the Material index as the parameter, to report the intersection and what Material should be used on this part of the geometry.

Attribute Variables may only be set in between these two function calls. This ensures that the Variables always represent the values of the closest hit yet found. If the Variables are read inside the Closest Hit Program it is guaranteed, that they represent the values of the closest intersection.

Typically Attribute Variables are used to communicate intersection specific information to the Closest Hit or Any Hit Programs such as surface normal vectors or texture coordinates.

Bounding Box Program

Geometry objects also need a Bounding Box Program assigned. It must return an axis aligned bounding box that fully encloses the primitive at the primitive index given as an argument. It is used by the Acceleration Structures while traversing and for building the acceleration tree. While a fast implementation is desirable, the accuracy of the bounding box is also important to build good quality acceleration trees. Accurate means that the bounding box should be as small as possible while still fully enclosing the primitive.

Closest Hit Program

A Closest Hit Program may be assigned to a Material. It is called once the nearest primitive that intersects the ray was identified. Typically it is used to implement a surface shader that calculates the result of the current ray. The intersection details such as texture coordinates should be communicated to the Closest Hit Program via Attribute Variables that were calculated in the intersection program. Then the Closest Hit Program can be used to calculate a color by doing texture lookups, calculating lighting and so on.

Further secondary rays may be recursively cast with `rtTrace`, to implement for example shadow effects or reflection. The results of the Closest Hit Program should be stored in the payload of the ray.

Any Hit Program

The Any Hit Program may also be assigned to a Material. It is called when an intersection program reports a potential intersection. This means it is called on any intersection that occurs, and as opposed to the Closest Hit Program, it might be called multiple times for a single ray cast. The intersections for which the program is executed may not be ordered along the ray, but eventually all intersections can be enumerated by calling `rtIgnoreIntersection` on each of them.

The Any Hit Program can be used if the application requires to perform actions at each surface intersection. Rays can also be terminated with `rtTerminateRay`, then the trace ends and no further executions of the Any Hit Program may be invoked. This can be used if only the knowledge whether the ray hits anything or not is needed, for example in shadow rays.

Miss Program

The Miss Program is directly assigned to the Context. It is invoked if a trace of a ray completely misses and finds no intersections. The Miss Program can be used to implement a background.

3 Related Work

VMD is an open source and easy to use and modify molecular modelling and visualization computer program. Its main purpose is the interactive display of molecular systems, such as proteins or nucleic acids. It was developed in the Theoretical and Computational Biophysics group at the Beckman Institute at the University of Illinois at Urbana–Champaign and was released in 1995 [HDS96]. Initially VMD was developed for Silicon Graphics workstations but since then it is under active development and has been ported to many other operating systems including various Unix systems and the Microsoft Windows platform, by offering a full-featured OpenGL version.

Ray tracing has been used for volume visualization for a long time and many works have been contributed to this. As hardware developed, it became possible to take volumetric ray tracing to interactive frame rates. Already 1991 Parker et al. present a brute-force ray tracing system for interactive volume visualization [PPL+99]. They use a shared-memory multiprocessor machine to achieve several frames per second, without using any of the graphic capabilities. As the development manager, Parker goes on to become a driving force behind the Nvidia OptiX ray tracing framework that takes the ray tracing to the GPU to open up for better performance. In [PBD+10], Parker et al. first introduce to OptiX and describe the various design decisions taken. With this framework it now becomes possible to gain interactive frame rates for volumetric ray tracing even on standard desktop PCs.

One of the biggest challenges for volumetric ray tracing is the physically correct lighting. As light can bounce around in a scene almost indefinitely, local lighting approaches are used. Lighting on surface geometry is comparatively straight forward and cheap, using for example the phong reflection model [Pho75]. Phong uses a combination of ambient lighting, diffuse reflection using Lambert's cosine law, and specular reflection. For volumetric data there are no clearly defined surfaces, so new lighting models are needed. Behrens et al. were able to to add shadows to a texture-based volume renderer with interactive frame rates [BR98]. The presented algorithm works without lighting calculations and instead uses pre calculated shadow maps to give the image a shaded appearance. Translucency introduces another large shading problem as the light traveling through semi transparent objects gets complexly scattered. In [KPHE02], Kniss et al. present a shading model that produces a qualitative appearance of translucency

by capturing volumetric light attenuation effects, that runs at interactive frame rates. An advanced lighting technique using a local approximation of ambient occlusion was introduced by Hernell et al. [HLY10]. It uses many acceleration techniques to achieve interactive frame rates with very good image quality.

4 Implementation

4.1 Project Setup

The Nvidia OptiX library serves as my starting point in this thesis and takes care of most of the ray tracing tasks. To get started with Nvidia OptiX, one needs Nvidia CUDA installed. Only certain Versions of CUDA are supported. I used OptiX 3.8.0 along with CUDA 7.0. With this combination of versions, Visual Studio 2010 through 2013 is supported. I used Visual Studio 2013. All my code is written in C++ using the Visual C++ 12.0 Compiler with all Language Extensions and Common Language Runtime support deactivated to stay true to the ISO C++11 standard. OptiX ships with some sample projects, so starting out with one of these samples seems to be a great idea. But the samples delivered with OptiX have strong inter-dependecies, that is, multiple sample projects have to be imprted to get one of them working. They also use many statically linked libraries that you have to take care, and a library made for the samples that ships with OptiX called sutil with many dependencies as well pollutes the project with a lot of code overhead. For building the samples CMake is used, which is not optimal for use with Visual Studio, because the automated build system of Visual Studio is preferable. In conclusion starting out with an empty project rather then one the samples is a faster and cleaner way to start with. In an empty project the CUDA and OptiX Include Directories must be included and the Optix Libraris optix.1.lib and optixu.1.lib must be added to the dependencies. The CUDA compiler must be added under 'Build Customizations' and configured to compile all the .cu files used for OptiX. The compile flags -ptx and –machine 64 need to be set to make the output files work with OptiX. From there the project should be fully set to work with OptiX.

4.2 Project Concept

Because this thesis has to later be integrated into VMD, it was important to me that the code is modular, clearly structured and easy to read, so that it can be integrated into a different software without trouble.

4.2.1 Host Side

The host side of the project is divided into two files. The main file called `main.cpp` sets up my testing environment by starting a window and providing functionality for inputs and fps display. This serves as a simulation of an environment that can be easily plugged out because all the rest of the Thesis is implemented in the second file called `setup_optix.cpp`. This file offers a compact interface that can be used in any environment. It offers the following functions:

setupOptix
: Handles everything to initialize the OptX context with specified width and height.

loadVolumeOptix
: Adds the volume from a specified file to the scene. This Function might be called multiple times to add more volumes.

loadTriangleMesh
: Adds an OBJ triangle mesh from the specified file to the scene. Again this file might be called multiple times to add more meshes.

compileOptix
: After all the geometries are added to the scene, this function handles the validation and compilation of the previously initialized OptiX Context to build the Acceleration Structures used in the raytracing.

renderOptix
: Renders the previously initialized and compiled context and returns an OptiX handle to the output buffer, that can be read in the unsigned byte BGRA format. Can be called multiple times to rerender the scene.

setCameraMatrix
: Between any of the renderOptix calls you can change the camera with this method by providing a transformation matrix.

All of the host side code is written in ISO C++11 that the project can be compiled with any compiler that might be used to compile the software the project will later be integrated in. This interface can be integrated and used in just about any environment. Besides these two files there is one more file called `helpers.h` that is used to implement small helper functions.

4.2.2 Device Side

On the device side, OptiX uses the Nvidia language CUDA C. The project is divided into the following files:

box.cu
Implements the intersection and Bounding Box Program for an axis aligned box. This geometry is used for volumes in this project.

raymarch_shader.cu
Implements the Closest Hit and Any Hit Program for the Material that is used on volumes. This is the core file for the ray march algorithm.

triangle_mesh.cu
Implements the intersection and Bounding Box Program for triangle meshes.

obj_material.cu
Implements the Closest Hit and Any Hit Programs for the Material used on triangle meshes.

pinhole_camera.cu
Implements the Ray Generation Program used in this project.

constantbg.cu
Implements the Miss and Exception Program used in this project.

optix_commonStructs.h
Defines some structs that are used throughout multiple files.

optix_helpers.h
Defines some helper function that are used throughout multiple files.

optix_phong.h
Implements the phong lighting used in the obj Material to clean up the code in that file.

4.2.3 Used Libraries

For loading the OBJ files I use the implementation in `OptixMesh.h` from the Sutil Library delivered with the OptiX samples. This library also comes with many dependencies, but once a different OBJ loader is used, the Sutil Library is no longer needed. For testing my project I use freeglut [1], but as the project will be extracted from my environment,

[1] http://freeglut.sourceforge.net/

freeglut will not be needed. Additionally I use the C++ standard library and Nvidia OptiX.

4.3 Display Environment

To test my project I needed a way to display the results. While still images give a good first impression, an interactive environment is invaluable for debugging purposes. Being able to quickly change the camera angle often helps to get insight on the errors happening. Because this environment does not add real value to the project it was more important that it is simple and fast to create rather than being sophisticated.

The OpenGL library freeglut was a good fit for this purpose, because it offers a slim easy to use and fast to learn interface for creating a window and displaying buffers on the screen. I added functionalities to change the camera angle with the left mouse button, pan with the right mouse button and zoom with the mouse wheel.

I set it up to continuously render the scene and added a fps counter (frames per second) to be able to measure the performance. The size of the window can be changed in the code to test different resolutions.

4.4 Implementation of Volumetric Ray Tracing

The high level idea for the volumetric ray tracing was to use a simple axis aligned cube in the ray tracing stage like in [KW03]. Once the cube is hit, the marching shader is called, which steps trough the volume along the ray and accumulates color samples.

To initially cast the rays that come from the camera, the Ray Generation Program uses the function: $U * x + V * y + W$ to calculate the direction of the ray coming from the pixel at (x, y). The typical U, V, W vectors are calculated in beforehand with the position, the looking direction and the up vector. The position of the camera is fixed to the position $(0, 0, 5)$. Camera movements are instead realized by using a transformation that simulates the camera.

The cube has always the size of one unit and is at the position (0,0). This simplifies the marching, because the position inside the cube can be directly mapped to the position in the 3D texture. Looking at how the graph is set up, clarifies that this is no restriction. In Fig. 4.1 you can see the graph that I use. Note that the Programs and Acceleration Structures have been omitted for clarity. Every geometric object has its own

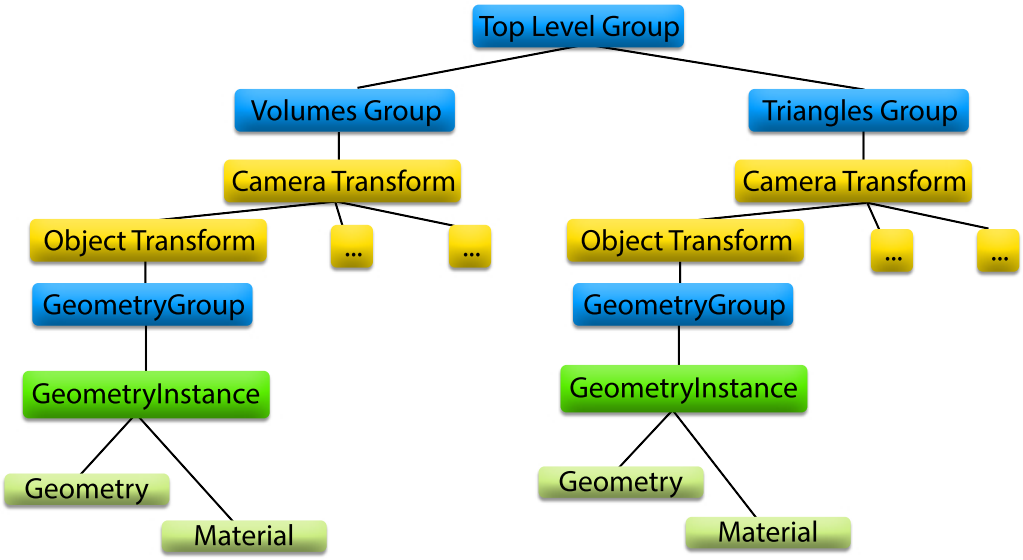

Figure 4.1: Optix Graph with separate top levels for the triangle meshes and the volumes

transformation that acts as the position and size of the object. This way the boxes can be freely sized, positioned and rotated.

Using axis aligned cubes also allows for easier and faster Intersection Programs. I use an efficient branchless ray-box-intersection algorithm based on the Slab method from [Owe98]. A special case to mention is when the camera is inside the volume. This produces a negative t-value for the front hit, which is not in the valid range and the volume will be completely ignored. Also the ray marching is supposed to start from the point where the camera is, as the part of the volume behind the camera should not be rendered. To fix this I just shift the front t-value to a small positive value, if it was negative before. This means it gets shifted right in front of the camera and the ray marching automatically works fine since it calculates the starting point from the changed t-value.

The Bounding Box Program of axis aligned boxes is equivalent to the size, so no calculations are needed. Via Attribute Variables I communicate the parametric t-values to the Material to identify the range in which the ray marcher needs to step. It is important to realize, that in the Intersection Program the ray is transformed in object space. OptiX normalizes the ray after the transformation to object space, which means that scale transformations are omitted. Because of this the calculated t-values are invalid and need to be transformed accordingly, to factor the scale transformations in. OptiX itself expects the wrong t-values for identifying the closest hit, so they must be transformed internally for the testing.

The ray marching is implemented as the Closest Hit Program of the cubes. With the previously calculated t-values, it is possible to identify the start and the end point of the ray intersecting the volume. At this point the step size can be calculated depending on the angle of the ray intersecting the volume and the resolution of the 3D texture. Then the algorithm steps through the volume, accumulating samples from the 3D texture and looking up the color values from the transformation function. The textures are stored in an OptiX Texture Sampler which offers efficient hardware interpolation. All color values are blended using the over operator with pre-multiplied alpha values seen in 4.2. If the marcher steps outside the volume or the current sum of blended colors is already fully opaque, the marching stops. The result of the color and alpha value is then stored in the payload.

To also support triangle meshes I included parts of the Nvidia OptiX samples. The Intersection Program uses the branchless triangle intersection algorithm from the math library delivered with Optix. The texture coordinates and the normal vector are calculated and communicated via Attribute Variables to the Material. For the Bounding Box Program the highest and lowest x and y values that appear in the triangle coordinates are used as the bounding box. To load an OBJ file to a triangle mesh, I use the loader from the Sutil Library from the OptiX samples. The Material for the triangle meshes implement Phong shading in the Closest Hit Program. It uses the normal vector and the texture coordinates to calculate the Phong lighting. Shadows are achieved by sending a ray in the direction of the light source. Via an Any Hit Program in the Material for the triangle mesh, it is tested whether anything is in between the point and the light source, and the ray is terminated immediately if any hit was detected. This implements binary shadowing.

4.5 Triangle Volume Intersection

The ray marching originally steps through the volume without testing for triangles. That means that all geometry inside of a volume is omitted and not displayed.

Testing for triangles by doing a full ray trace every step would be too much overhead. Instead I test once for the closest triangle in the volume and store the t-value of the distance and the calculated color. To do so I had to deploy another ray type, for which the triangle Material would store the t-value as a result in the payload. At this stage it would be possible to incorporate this into the original ray type just by adding the t-value to the payload, but when talking about transparency in section 4.6 it becomes clear that the two types fulfill different tasks. The new ray type only searches for the closest triangle and stores its color, while usually if the triangles are semi-transparent you would blend all triangles along the ray together for the result. Also, only triangles

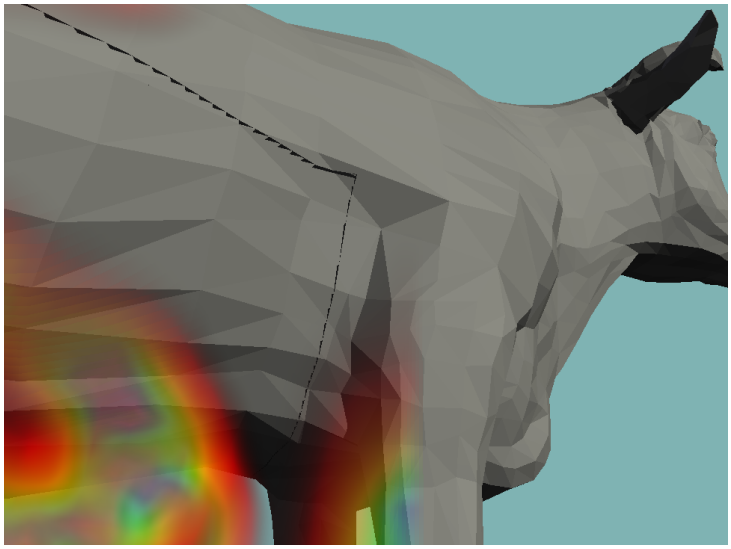

Figure 4.2: Artifacts caused by not blending the color of a triangle due to the step size

should be found, as intersecting volumes are not supported, because that would require a completely different approach. For intersecting volumes a ray marcher that can march multiple volumes at the same time would be needed. Because every volume has its own Material and therefore its own marcher, the current approach can not support this. To only hit triangles and no volumes the graph in Fig. 4.1 shows that I have two separate Groups, one for all volumes and one for all triangles. This way I can give just the triangle group to the ray tracer to find only triangles. Because of this separation the camera transformation needs to exist two times, it is important that they always stay the same.

The valid range for this closest triangle trace is set to the end of the volume, so no triangles outside the volume will be found. With the t-value and color results of the closest triangle found, the ray marching now steps through the volume until the t-value of the triangle is reached. Then the color of the triangle is alpha blended using the over operator 4.2 and the marching stops if full opaqueness is reached, which is always true for opaque triangle geometry. For the support of semi-transparent geometry inside a volume read section 4.6.

A new Miss Program sets a miss flag in the payload for the new ray type, to easily determine in the ray marching, whether a triangle was hit inside the volume or not. If a triangle is found inside the distance of the very last step, it would not be blended because the marching stops as it marched outside the volume. This is especially apparent for geometry that penetrates the bounds of the volume cube and can be seen in Fig. 4.2. To avoid this you have to check whether the closest triangle found would still be inside the volume and blend it to the result before stopping the marching.

4.6 Transparency

To support semi-transparent triangle geometry I adjusted the Closest Hit Program in the Material for the triangle meshes in the way that it does not just write the resulting color into the payload, and instead uses alpha compositing [PD84] to blend it to the color already inside the payload. This requires to extend the payload, to also hold the alpha value and that the color values are initialized to zero. If after the blending the color is still not fully opaque a new recursive ray is spawned just behind the current triangle to find the next triangle in line. Since the color is first blended and then the new ray is spawned, front to back blending via the over operator is used:

$$C_o = \frac{C_a\alpha_a + C_b\alpha_b(1-\alpha_a)}{\alpha_a + \alpha_b(1-\alpha_a)} \tag{4.1}$$

With the output color C_o, the front color C_a, the back color C_b and their respective alpha values α_a and α_b.

All colors in my project are handled as if they are pre-multiplied by their alpha values. This allows for a more efficient alpha compositing with the simplified form:

$$\begin{aligned} c_o &= c_a + c_b(1-\alpha_a) \\ \alpha_o &= \alpha_a + \alpha_b(1-\alpha_a) \end{aligned} \tag{4.2}$$

With the c_o, c_a, c_b being the the C_o, C_a,C_b with their alpha values pre-multiplied and α_o the alpha value of the output color.

The transparency of volumes is achieved the exact same way. After finishing the ray marching, blend the result to the color in the payload via 4.2 and cast an new ray just behind the volume.

A special case is when triangle geometry is inside of a volume, since a different approach is used to render them as described in section 4.5. If a semi-transparent triangle is found as the closest triangle and the marcher gets to the point where it blends the triangle, the marcher does not just stop and instead searches for the next closest triangle and continues. To find the next triangle the valid range of the ray is set to the range between the current point in the marching and the end of the volume. Because the next triangle is searched after the marcher steps over the previous triangle, the next in line will be found instead of the same one again. After this the marching continues as normal until it steps outside the volume, full opaqueness is reached, or the current closest triangle needs to be blended.

4.7 Ambient Occlusion

To achieve better image quality for the volumetric ray marching, the advanced lighting technique ambient occlusion is used [HLY10]. Ambient occlusion is a method to approximate how bright light should be shining on any specific part of a surface, based on the light and it's environment. It is mused often calculated by casting rays in every direction from the surface. Rays which reach a light source increase the brightness of the surface, whereas a ray which hits any other object contributes no illumination. Instead of this global approach I use a local approximation that accumulates how much density is found in a local sphere.

So far, the color of the point in space the marcher encounters was only determined by the color of the 3D texture and the transformation function. Ambient occlusion determines the light intensity of this point by approximating how much ambient light could possibly reach this point. Sample rays are sent in random directions to test how dens the surrounding in a predefined sphere is. To do so, all these rays march through the volume just as the ray marcher does and accumulates the values read on each step. Instead of accumulating the color, they only accumulate the alpha value read from the transformation function. The marching is only continued for a specific amount of steps to sample only points inside the sphere. To also identify triangles the same method is form section 4.5 is used. Then the results of the ambient occlusion rays are averaged to approximate how much the point is shadowed. To do the shadowing I use alpha compositing to blend black in front of the color at the current point. For the alpha value the result determined by the ambient occlusion is used.

4.8 Instructions to add more Geometry Types

For this project I implemented only triangle meshes as geometry besides the volumes. In the future, further types of geometry might be desired. I will quickly describe all the steps needed to add those to the project. First the geometry has to implement the Intersection and Bounding Box Programs, and the Material has to implement the Closest Hit Program using my payload defined in `optix_commonStructs.h`.

The Geometry Instance can be added to a new Group under the top level Group seen in Fig. 4.1. In this case the newly added Group should also add a copy of the Camera Transformation which also needs to be updated every time the camera changes via `setCameraMatrix`. To make the geometry also show up inside of volumes they should be added to the already existing Triangles Group, that can be seen as a Non-Volumes Group at this point. Then the Material must also implement the Closest Hit Program for the

second ray type that returns the t-value and the color of the closest object as described in section 4.5. If it should also be used to shadow triangles and other geometry, then an Any Hit Program for the shadow ray type should be implemented as well.

5 Results

5.1 Visual Results

In Fig. 5.1 you can see a snapshot with two intersecting volumes represented by the bucky ball. A triangle mesh in the form of a gray, semi-transparent cow is partially inside one of the volumes. The bucky balls are displayed correctly and a transformation function gives it colorization. The intersection of the volumes cuts a portion of the second volume since intersecting volumes are not supported. The cow inside the first volume is being rendered as expected and the inside parts comply with the outside parts for a seamless transition. When zoomed in very far, a gap of one pixel may be identified between the inside and the outside part. That is because the new ray is casted behind the volume with a tiny distance so that the volume is not hit again. But zoom levels of this type are not expected. Both the volumes and the cow are transparent and geometry in the background is correctly displayed through the objects. Correct Lambert shading can be seen on the cow. The shadowing can be seen at the horn of the cow that correctly throws a shadow on the neck.

The effects of the ambient occlusion can be best seen on single color volumes (see Fig. 5.2). Without the ambient occlusion it is hard to identify the contours of the volumetric data and it looks very flat, almost like a 2D sprite. With ambient occlusion activated as in Fig. 5.3, the volume gains depth and looks and looks more three dimensional. This clearly improves the visual quality and helps to understand the shape of the bucky ball.

5.2 Performance

For measuring the performance I used the scene from Fig. 5.1. The volumes have a voxel resolution of 32^3 and the cows are triangle meshes consisting of 5805 triangles. My testing machine uses an AMD FX-6300 CPU with 16GB DDR3 RAM and the NVIDIA GeForce GTX 960 GPU. The scene is rendered in a resolution of 1024x768 pixels. The back volume and cow are only activated for certain measurements to test the effect of

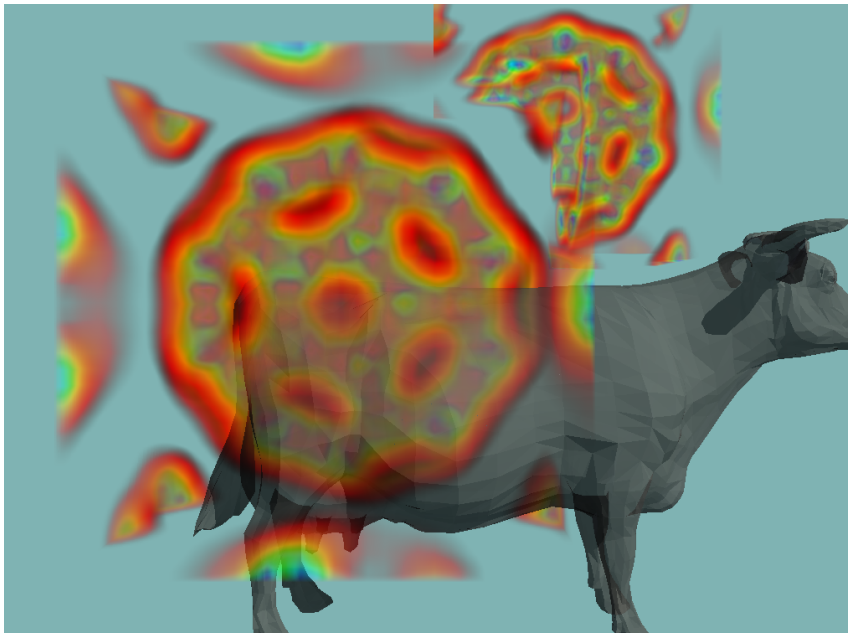

Figure 5.1: An image from my software showing two intersecting volumes and a triangle mesh. Because intersectiong volumes are not supported you can see the wrong rendering at the intersection.

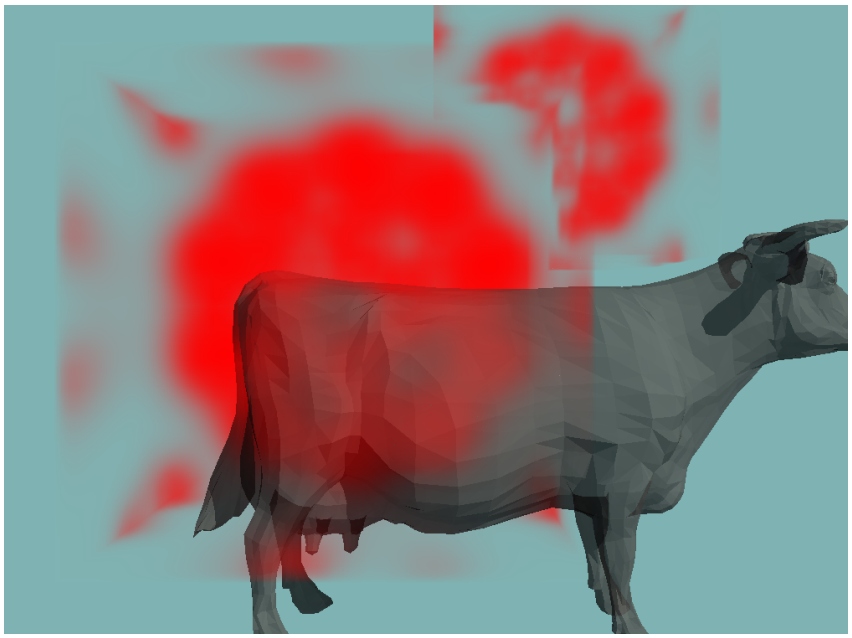

Figure 5.2: Using only a single color for the volume without ambient occlusion.

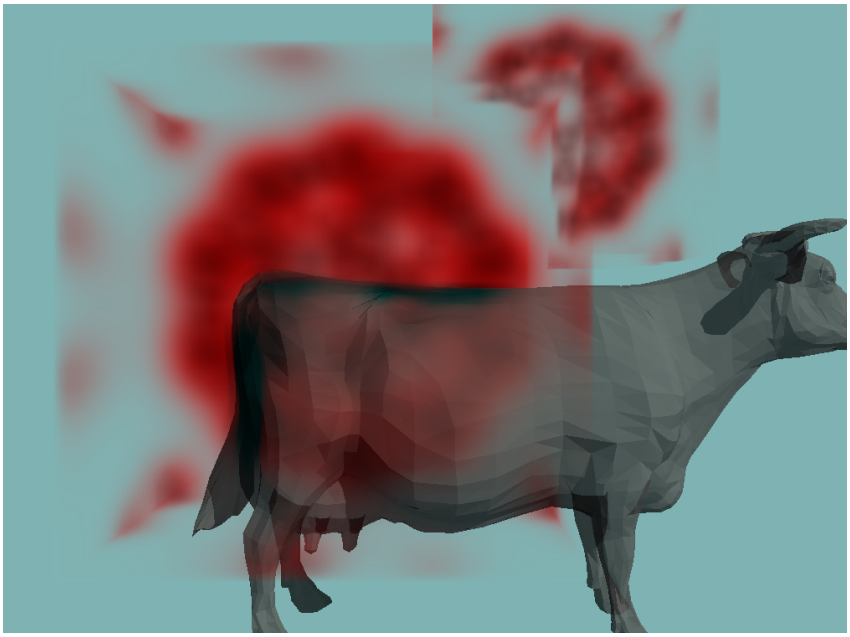

Figure 5.3: With ambient occlusion the volume looks more three-dimensional

increasing the objects in the scene. When only the front volume and cow is activated I will refer to it as the small scene. In table 5.1 you can see the results for the small scene with various visual features enabled. Without any of the visual features and fully opaque triangles the testing machine reaches 57.1 FPS. Using shadows for the triangle geometry, an additional shadow ray has to be cast for every hit. But since the ray is terminated on the first hit that is found, it is expected that only half of all triangles need to be checked. Therefore the loss in performance with only 7 FPS is rather low. Using semi transparent triangle geometry on the other hand costs more than 20 FPS. This is because for every ray that hits a triangle, another ray must be traced. Since the cow has a front and a back side, every ray hitting the cow hits two triangles and thus two additional rays are traced.

In the second table 5.2 the results for adding more geometry to the scene are enlisted. For these tests the triangles were set to be semi-transparent and use shadowing. The small scene is used as the base and depending on the test, the second cow or volume were added. It can be seen that adding double the amount of triangles has a huge performance impact, much larger than adding another volume. Increasing the amount of triangles increases the cost for every ray trace because a larger list of geometry objects has to be tested for intersection. That in turn increases the cost for the shadowing as well as the semi-transparency of the triangles, hence the large performance drop of 11.2 FPS. Since the second volume takes up only a small portion of the image, only few rays need to additionally do the ray marching. But still it shows that the ray marching actually is rather cheap. In fact completely removing both volumes from the scene only

No Effects	57.1 fps
Triangle Shadowing	50.1 fps
Transparency	36.5 fps

Table 5.1: Performance measurements for the small scene with the specified effect active

Small Scene	15.0 sec.
Two Volumes	15.5 sec.
Two Triangle Meshes	19.3 sec.
Both doubled	19.8 sec.

Table 5.3: Shows the effect of increased geometry count with ambient occlusion active.

Small Scene	30.1 fps
Two Volumes	28.3 fps
Two Triangle Meshes	18.9 fps
Both doubled	18.1 fps

Table 5.2: Shows the effect of increased geometry count. All effects but ambient occlusion are active.

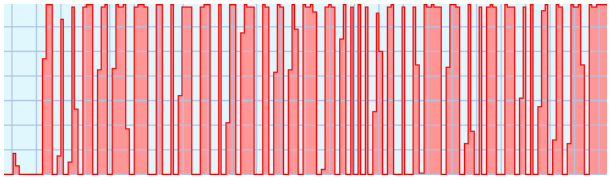

Figure 5.4: GPU load when rendering the scene with ambient occlusion

gives a performance improvement of 6 FPS over having both. Increasing the resolution intuitively increases the amount of rays cast by 4 times, the last test in table 5.2 shows exactly that and that the resolution of 2048x1536 is the maximum for interactivity.

The last table 5.3 shows the results for activated ambient occlusion. Ambient occlusion drops the performance tremendously, so that no interactivity is possible anymore. Adding the second volume almost has no performance cost. The reason for this can be seen in Fig. 5.4 which displays the GPU load. From the graph you can figure out that the GPU has many downtimes that could be caused by a stalling thread. Most likely this is because the ray marching is fully implemented in a single thread. The additional ambient occlusion runs completely on the same thread, stalling all others. Therefore adding another volume does not affect the performance since the unused processing units of the GPU are used for the additional rays that need to do the ray marching. This tells me that the performance of the ambient occlusion is not sufficient, since even running it on clusters will not help to get interactive frame rates. Better parallelization or an iterative approach could be used to achieve better results.

5.3 Limitations

As described in section 4.5 , the intersection of volumes can not be supported with the current method. This can be a limitation in certain situations where you want to overlay 3d data to achieve the wanted effect. Note that it is possible to combine the volumetric data in a preprocess step with rather low cost. Since only box shaped volumes are supported, depending on the positioning of the volumes a lot of empty space may be generated. To avoid overhead in these regions, empty space skipping like in [LMK03] can be used, to increase performance.

The Nvidia OptiX framework runs only on Nvidia GPUs. While in the area of research Nvidia is more commonly used, it is a limitation especially to the end user who will not be able to run the software on different hardware.

6 Conclusion & Outlook

6.1 Summary

The Nvidia OptiX framework shows that ray tracing can be done in real-time with the current hardware. It is a very well designed framework that offers a lot of freedom to the user. The OptiX Programming Guide [15a] is well written and helped me to learn the framework. The samples delivered with OptiX, also offer a good way of learning, but I do not recommend using one of the samples as starting point for your own project. I developed a method to use OptiX for volumetric ray tracing, which also can be combined with the common ray tracing for non volumetric geometry. I was able to solve the intersection problem for volume-triangle intersection and the transparency problem for both geometry types. Triangle meshes can be shadowed by other geometry and using ambient occlusion I increased the visual results. The ray tracer works in real-time and interactivity can be achieved on a standard desktop PC. With activated ambient occlusion the frame rates do not allow interactive frame rates and can therefore only be used for offline rendering.

6.2 Future Work

In the future this project will be integrated to the VMD software. To use the many different visualizations that VMD offers, possibly more geometry types then just triangle meshes will be needed. For this reason, I described all steps needed on how to add more geometry types in Sec. 4.8.

If the performance is not sufficient there are some acceleration methods that can be used. For example empty space skipping as explained in [LMK03] can speed up the ray marching by subdividing it into smaller areas so that the ray marcher can identify larger regions that are empty and can just skip them. One of the major performance issues with triangle geometry is that for every translucent triangle a new ray is being cast to search for the next triangle in the background. Since the first ray already collected all triangles that are hit by the ray to determine which one is the closest hit, it is possible to

store all the intersected triangles and do the sorting manually. This denies refraction of light on translucent surfaces which may be used to enhance image quality, but if no refraction is needed, this can potentially increase the performance dramatically. To achieve interactive frame rates even with activated ambient occlusion, an iterative design as in [HLY10] could be implemented that refines the ambient occlusion over multiple frames.

Bibliography

[15a] *OptiX Programming Guide*. Published by NVIDIA Corporation. Version 3.8. 2015. URL: http://docs.nvidia.com/gameworks/content/gameworkslibrary/optix/optix_programming_guide.htm (cit. on pp. 12, 37).

[15b] *OptiX QuickStart Guide*. Published by NVIDIA Corporation. Version 3.8. 2015. URL: http://docs.nvidia.com/gameworks/content/gameworkslibrary/optix/optix_quickstart.htm (cit. on p. 11).

[BR98] U. Behrens, R. Ratering. "Adding Shadows to a Texture-based Volume Renderer." In: *Proceedings of the 1998 IEEE Symposium on Volume Visualization*. VVS '98. Research Triangle Park, North Carolina, USA: ACM, 1998, pp. 39–46. URL: http://doi.acm.org/10.1145/288126.288149 (cit. on p. 19).

[Gla89] A. S. Glassner, ed. *An Introduction to Ray Tracing*. London, UK, UK: Academic Press Ltd., 1989 (cit. on p. 9).

[HDS96] W. Humphrey, A. Dalke, K. Schulten. "VMD – Visual Molecular Dynamics." In: *Journal of Molecular Graphics* 14 (1996), pp. 33–38 (cit. on pp. 7, 19).

[HLY10] F. Hernell, P. Ljung, A. Ynnerman. "Local Ambient Occlusion in Direct Volume Rendering." In: *IEEE Transactions on Visualization and Computer Graphics* 16.4 (July 2010), pp. 548–559 (cit. on pp. 20, 29, 38).

[Ize09] T. Ize. "Efficient Acceleration Structures for Ray Tracing Static and Dynamic Scenes." MA thesis. The University of Utah, 2009 (cit. on p. 11).

[KPHE02] J. Kniss, S. Premoze, C. Hansen, D. Ebert. "Interactive translucent volume rendering and procedural modeling." In: *Visualization, 2002. VIS 2002. IEEE*. Nov. 2002, pp. 109–116 (cit. on p. 19).

[Kuc87] R. Kuchkuda. "An Introduction ro Ray Tracing." In: (Dec. 1987) (cit. on p. 9).

[KW03] J. Krüger, R. Westermann. "Acceleration Techniques for GPU-based Volume Rendering." In: *Proceedings IEEE Visualization 2003*. 2003 (cit. on p. 24).

[LMK03] W. Li, K. Mueller, A. Kaufman. “Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering.” In: *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*. VIS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 42–. URL: http://dx.doi.org/10.1109/VISUAL.2003.1250388 (cit. on pp. 35, 37).

[Owe98] G. S. Owen. *Ray - Box Intersection*. 1998. URL: http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm (cit. on p. 25).

[Pag08] O. V. H. Page. *What is VMD?* 2008. URL: http://www.ks.uiuc.edu/Research/vmd/allversions/what_is_vmd.html (cit. on p. 7).

[PBD+10] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, M. Stich. “OptiX: A General Purpose Ray Tracing Engine.” In: *ACM Transactions on Graphics* (Aug. 2010) (cit. on pp. 11, 19).

[PD84] T. Porter, T. Duff. “Compositing Digital Images.” In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pp. 253–259. URL: http://doi.acm.org/10.1145/964965.808606 (cit. on p. 28).

[PH10] M. Pharr, G. Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann, 2010. URL: http://www.amazon.com/Physically-Based-Rendering-Second-Edition/dp/0123750792?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0123750792 (cit. on p. 10).

[Pho75] B. T. Phong. “Illumination for Computer Generated Pictures.” In: *Commun. ACM* 18.6 (June 1975), pp. 311–317. URL: http://doi.acm.org/10.1145/360825.360839 (cit. on p. 19).

[PPL+99] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, P. Shirley. “Interactive Ray Tracing for Volume Visualization.” In: *IEEE Transactions on Visualization and Computer Graphics* 5.3 (July 1999), pp. 238–250. URL: http://dx.doi.org/10.1109/2945.795215 (cit. on p. 19).

[WBS07] I. Wald, S. Boulos, P. Shirley. “Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies.” In: *ACM Trans. Graph.* 26.1 (Jan. 2007). URL: http://doi.acm.org/10.1145/1189762.1206075 (cit. on p. 11).

[Wei06] D. Weiskopf. *GPU-Based Interactive Visualization Techniques (Mathematics and Visualization)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006 (cit. on p. 11).

All links were last followed on February 28, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature